INTEGRATING FORMAL METHODS INTO SOFTWARE DEPENDABILITY ANALYSIS

John C. Knight Luís G. Nakano

(knight | nakano)@virginia.edu

Department of Computer Science
University of Virginia
Charlottesville, VA 22903-2442, USA

An abstract submitted to:

The Twenty-Third Goddard Software Engineering Laboratory Workshop

Contact author:

John C. Knight

Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903-2442, USA

knight@virginia.edu
+1 804 982 2216 (Voice)
+1 804 982 2214 (FAX)

INTEGRATING FORMAL METHODS INTO SOFTWARE DEPENDABILITY ANALYSIS

John C. Knight
Department of Computer Science
University of Virginia

Luís G. Nakano
Department of Computer Science
University of Virginia

1. Introduction

Formal methods are techniques based in mathematics that facilitate the precise specification and verification of software systems. Their use has been demonstrated in a number of experiments and industrial development projects [3]. Despite these demonstrations, formal techniques remain the exception rather than the rule in system development. One of the issues raised about the use of formal methods is the lack of any means whereby their results can be used in the broader context of system dependability analysis, i.e., the analysis of a complete system including hardware and software. For example, what would be the benefit at the *system* level of the use of a formal specification in the preparation of the system *software*?

The issues that we address in the work summarized here are:

- For what parts of a complex software system should formal methods be used?
- How can the results of formal analysis be used in the overall dependability analysis of the entire system?

We summarize a process by which these issues are addressed, and show thereby how to determine the role of formal methods in any particular development and how to exploit the results of formal analysis in system dependability analysis. At the workshop, we will illustrate the process using examples from analysis performed on parts of the design of an experimental nuclear-reactor control system.

2. Dependability Analysis

Analysis of the dependability of safety-critical systems is essential in order to provide estimates of the expected losses (life and/or property) that such systems will cause per unit of operating time, i.e., their risks. These risk estimates are used by developers, users, policy makers and others to make informed decisions about deploying safety-critical systems based on the expected benefits and losses to society.

Risk analysis has not been applied as successfully to software-based safety-critical systems as it has to hardware-only systems. The reason is the discrete nature of software—it causes complexity not usually found in analog hardware and prevents interpolation of test results commonly applied to hardware-only systems. The result is a situation in which the hardware elements of a system are typically analyzed in depth but software is handled in only a very limited way, often as a “black box”.

Life testing is an approach to software dependability assessment in which the software is treated as a black box. The software is executed continuously in its operating environment for a period of time proportional to the duration of the mission and inversely proportional to the acceptable probability of failure. Unfortunately, it has been shown [2] that life testing is not a feasible approach to the dependability assessment of life-critical software because the duration of testing required is excessive. To reduce the need for testing, reliability growth models have also been tried [1]. By modeling the development of software in terms of testing and fault removal, it is argued that an estimate for software reliability can be obtained with lower test requirements. If it works at all, this approach only works for modest levels of dependability.

Formal methods are often advocated as an approach to developing dependable software. But poor tool support, the complexity of the systems, and the difficulty of using the techniques have limited the application of formal methods in many cases. The application of formal methods just to the safety-critical parts of a system is a valid approach, but it requires that the safety-critical parts be identified and delimited. No general technique for isolating the safety-critical components of systems is available, however. In addition, it is not clear how to determine the properties of a system (or part of one) that are relevant to its safety. Again, no technique so far has been widely accepted, and most applications of formal methods try to establish properties chosen in a non-rigorous manner. Though clearly useful, this utility is informal—such properties do not contribute formally to the overall *system* dependability analysis. In summary, though formal methods are of value, it is not clear how they should be applied nor how to use the fact that they have been applied in system dependability analysis.

Given this situation, an integrated approach that: (a) addresses both the software and hardware elements of a system; and (b) exploits the tremendous potential of formal methods is needed. In this paper, a comprehensive approach to system dependability analysis based on traditional techniques for risk analysis is summarized. The approach models software as a set of interacting components based on the structure of the software. By viewing software this way, software analysis can be integrated fully into the models used presently for hardware. The resulting composite models provide details of those conditions in which hazards might occur as a result of erroneous software operation thereby identifying precisely where attention to software dependability must be focused. As such, these conditions can be the target of formal analysis so that confidence is gained about the right properties of the right parts of a software system.

3. A Component Model of Software Dependability

Both simple life testing and reliability growth models ignore the structure of software when obtaining estimates of software failure rates thereby requiring either that impossibly large numbers of tests be performed or that failures induced in one component by another be ignored. Unfortunately, however, if one appeals to formal methods as an alternative approach, one is faced with the fact that formal methods do not provide stochastic estimates and so cannot be used easily in place of testing. And, as we have already noted, it is not possible to identify precisely where or how such methods can be applied effectively to just parts of large systems.

Traditional dependability analysis techniques, such as fault-tree analysis and failure-modes-and-effects analysis, are performed for hardware-only systems using complete knowledge of the internal design. Typically, the software in software-based systems cannot be analyzed this way because the interactions between components have either been ignored or not obtained

rigorously. Clearly, software components such as functions and tasks interact extensively, but this is not to be the case (or is assumed not to be the case) in archetypal hardware-only systems.

Since techniques that model software as a monolithic entity have not achieved sufficient fidelity, we have developed an approach in which software is modeled as a graph with components as nodes and interactions as edges. An event associated with the failure of a software component then appears as a separate entity in the system fault tree. However, traditional *quantitative* analysis cannot be undertaken without further analysis because of the component interactions. *Qualitative* analysis, however, is possible and the comprehensive system fault tree allows those parts of the software whose failure might lead to a hazard to be identified easily.

In our approach, interactions are determined based on a component-interaction model and then minimized using architectural techniques. The resulting fault tree is then analyzed quantitatively using extensions to fault tree analysis that include dependencies [5].

Of critical importance is the fact that the failures of individual *software* components now appear in the *system* fault tree. This permits system design decisions to be taken to reduce vulnerabilities, but, more importantly, it indicates what aspects of the software will benefit most from the use of formal methods and how. For example, if a software component is deemed to be critical because the fault tree shows that its failure would lead to a hazard with unacceptable probability, then the component can be subjected to detailed formal analysis. If it can be shown to be correct via proof, then its probability of failure can be assumed to be close to zero and increased confidence gained in the system's safety. The role of formal methods is then clear.

4. Component Interaction Model

In developing an analysis-by-components approach to modeling software, the first step is to determine how one software component can affect another. There are, of course, a multitude of ways that this can occur, but there is no basis in either models of computation or programming language semantics for performing a comprehensive analysis.

We chose to approach this problem by viewing component interaction as a *hazard* and basing our analysis on a fault tree for this hazard. In this way, we have documented, albeit informally but rigorously, all possible sources of software component interaction. The fault tree is quite large and we cannot include it here in detail. The events in the fault tree are based on the semantics of a typical procedural programming language, and the results apply to all common implementation languages such as Fortran and C.

In order to reflect the syntactic structure of procedural languages accurately, we define the term component to mean either (a) a function in the sense of a function in C, (b) a class in the sense of a class in C++, or (c) a process in the sense of a task in Ada. We make no assumptions about how components can provide services to each other (in a client/server model) or collaborate with each other (in a concurrent model) or otherwise interact.

As an example of the interaction model, figure 1 shows the top of the component-interaction fault tree. With no loss of generality, in this fault tree we consider only two components because there can be no interaction between components if there is no pair-wise interaction. Since information flow between A and B is symmetric, only one of the cases need be considered.

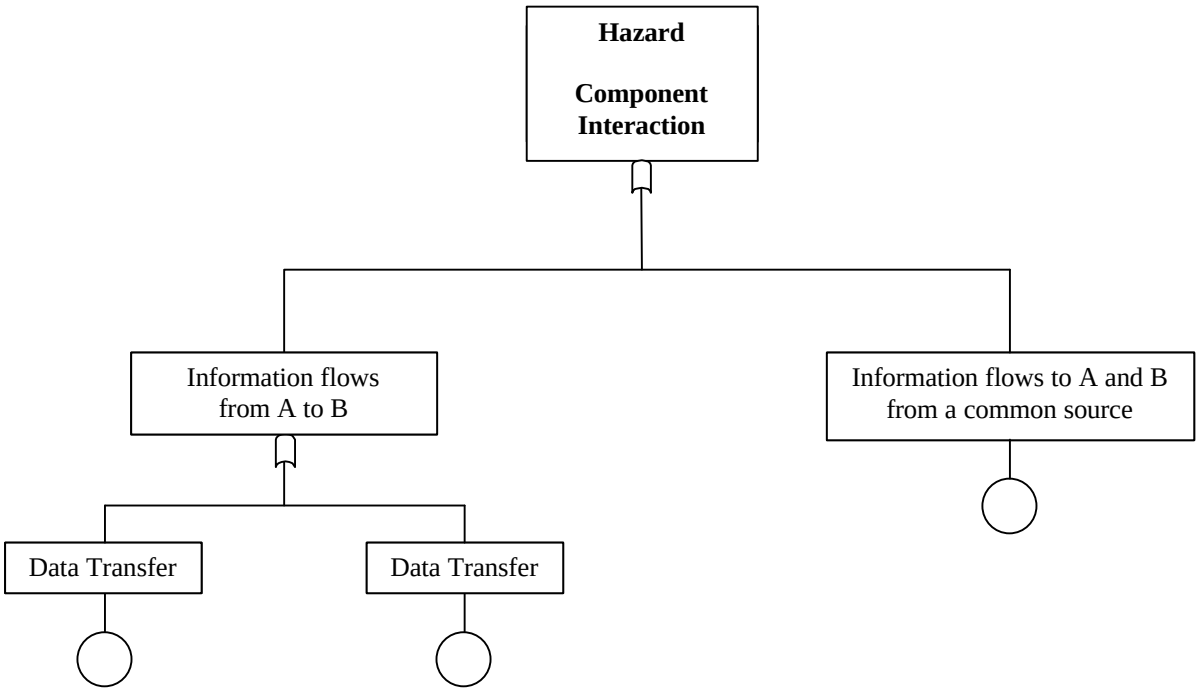

Figure 1: Fragment of component interaction fault tree

In the first level of the partial tree shown in figure 1, component interaction can be caused by information flow from A to B or by a common source of information. Thus, these are the two events shown. Note that component interaction does not necessarily mean *intended communication* in any format. Rather, it includes both intended and non-intended interaction between components. In addition, information flow does not mean just transfer of data. Flow of control is also information in the sense of the analysis that we wish to perform.

The complete interaction model derives sources of interaction in all semantic areas including shared data, memory management (e.g., one task consuming all memory thereby causing others to fail), task communications (e.g., priority inversion and deadlock), and exception generation and propagation.

5. Design Techniques for Analyzability

If analysis using our software component model is to be complete, it is essential that interactions between components that have to be analyzed always be detectable. Analysis of the component interaction model indicates that several potential causes for unwanted interaction cannot be discovered by static analysis of the system. Dynamic scheduling of functions and dynamic resource allocation, for example, have the potential for leading to failure under circumstances that are unpredictable. Similarly, other characteristics of software designs have the potential for increasing the complexity of the analysis or even making it infeasible.

Analytic feasibility requires that these sources of interaction be eliminated and this requires that certain restrictions be imposed. Both imposing the restriction and showing that a system meets them is best achieved by explicit use of design choices, for example:

- All resources must be statically allocated.
- All scheduling actions must be static.
- Execution times of components must be bounded.
- Inter-task communications must be synchronous.

This list, although not exhaustive, illustrates the properties that were derived from the component interaction model. Provided the complete set of design restrictions is met, all interactions between components of a software system can be analyzed. Achieving analytic feasibility of complex software systems using architectural techniques such as these is not unique to the approach we have developed. The SAFEBus architecture [4], for example, used in the Boeing 777 air transport enforces several of these properties to facilitate the safety analysis of the final system.

6. Quantitative Analysis

The final step in the approach that we have developed is quantitative analysis of complete systems including both hardware and software. The composite fault tree contains nodes describing failure events of all system components and all interactions between components are known. To complete the part of the quantitative analysis associated with the software nodes, we have developed an extension to the cut-set technique employed with conventional fault trees. The extension, termed *hazard-causing sequences*, involves enumerating all sequences of software component failures that could cause a hazard and analyzing each such sequence to show that its probability of occurrence is sufficiently small. If this analysis reveals a sequence whose probability of occurrence is not sufficiently small, formal techniques (perhaps combined with certain restricted forms of testing) can be applied to the sequence in order to either reduce the probability to a sufficiently small value or to show how the system design can be modified to make the associated sequence less critical.

7. Summary

In order to better model the dependability of complex software-based systems, we have developed an approach that uses the design of the software (viewed as a set of interacting components) as a basis for analysis. This approach permits the critical elements of the software to be identified and subjected to analysis using formal techniques. The approach, therefore, permits a clear determination to be made of the most appropriate application of formal methods to a large system and permits the results of formal analysis to be included in comprehensive system dependability analysis.

At the workshop we will describe the approach in detail, present the complete component interaction model, discuss the analytic techniques used in analysis of the composite fault-tree model, and illustrate the approach using analysis performed on parts of the design of an experimental nuclear reactor control system.

References

1. Brocklehurst, S.; Littlewood, B. *Techniques for prediction analysis and recalibration.* Chapter 4. In: Lyu, M. R., (ed). **Handbook of Software Reliability Engineering.** IEEE Computer Society, Los Alamitos, CA, 1995.
2. Butler, R. W.; Finelli, G. B. *The infeasibility of quantifying the reliability of life-critical real-time software.* In: **IEEE Transactions on Software Engineering**, v. 19, n. 1, p. 3–12, Jan. 1991.
3. Craigen, D; Gerhart S; and Ralston, T. *An international survey of industrial applications of formal methods*. National Institute of Standards Technology, U.S. Department of Commerce, (March 1993)
4. Hoyme, K.; Driscoll, K. *SAFEbus.* In: **Proceedings of the 1992 IEEE/AIAA 11**th. Digital Avionics Systems Conference, Seattle, WA, USA, 5Ð8 Oct. 1992, p. 610, 68–73. IEEE, New York, NY, USA, 1992.
5. Pullum, L. L.; Dugan, J. B. *Fault tree models for the analysis of complex computer-based systems*. In: **Annual Reliability and Maintainability Symposium. 1996 Proceedings. The International Symposium on Product Quality and Integrity**, Las Vegas, NV, USA, 22Ð25 Jan. 1996, p. 200–7, 1996.